Embedding it better...

µTasker Document

µTasker – Hardware Timers

Table of Contents

1. Introduction

Processors generally contain a number of timers. These are used, for example, to generate periodic interrupts, delays, frequencies or pulse-width-modulation signals; for counting external events or measuring periods of external signals.

The capabilities and use of such timers can vary greatly depending on the processor type.

This document describes the timer interface in the µTasker project which aids in simple control of such timers is a generic manner. Much of the timers' capabilities can also be simulate in the µTasker simulator, making the verification of new configurations and timer behaviour possible in user projects.

2. Timer Control Interface

The µTasker project uses a common interface for control of various interrupt capable peripherals.

```
fnConfigureInterrupt(*void)
```

The timer control is a particular case of using this interface and its use will be further detailed in the following sections.

In order to use the hardware timer support the specific hardware module(s) in the processor should first be activated. See appendix A for a complete list of timer modules supported in various processor packages.

Most processor types have a general purpose timer module which is activated in `app_hw_xxxx.h` (where `xxxx` is the processor type) by the define `SUPPORT_TIMER` (or similar).

3. Configuring a Single-Shot Time Delay

```
static void fnConfigure_Timer(void)
{
    static TIMER_INTERRUPT_SETUP timer_setup = {0}; // interrupt configuration parameters

    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.int_priority = PRIORITY_TIMERS;
    timer_setup.int_handler = timer_int;
    timer_setup.timer_reference = 2;                // timer channel 2
    timer_setup.timer_mode = (TIMER_SINGLE_SHOT | TIMER_US_VALUE);  // single shot us timer
    timer_setup.timer_value = 100;                  // 100µ delay
    fnConfigureInterrupt((void *)&timer_setup);     // enter interrupt and start timer
}
```

This example (*for Luminary Micro devices*) shows a timer being configured to generate an interrupt after a delay of 100µs. It uses a general purpose timer, whereby channel 2 is used for the delay. When the single-shot timer fires the interrupt call-back `timer_int(void)` is called from within the timer interrupt routine.

```
static void timer_int(void)
{
    TOGGLE_TEST_OUTPUT();
    fnConfigure_Timer();
}
```

This example interrupt routine is toggling an output (for visibility) and restarting a further single-shot hardware timer.

Note that the user interrupt handler doesn't need to reset any hardware flags since the driver interrupt handler is responsible for this work. The user must however be aware that the code is running in a sub-routine to the timer interrupt handler and so should generally be kept as short as possible. It is typical for such routines to send an event to a task so that extra work can be triggered (eg. `fnInterruptMessage(OWN_TASK, TIMEDELAY_1);`).

The timer module will generally be set to low power mode (power down or similar) after a single-shot timer has fired, in order to optimise power requirements when the timer is no longer in use.

See Appendix B for further examples of generating a single-shot interrupt delay for various processor types and using various timer modules in the processors.

4. Configuring a Periodic Interrupt

Periodic interrupt can be configured by using the same interface as for single-shot interrupts. Rather than setting the single shot mode a period mode is set.

```
static void fnConfigure_Timer(void)
{
    static TIMER_INTERRUPT_SETUP timer_setup = {0}; // interrupt configuration parameters

    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.int_priority = PRIORITY_TIMERS;
    timer_setup.int_handler = timer_int;
    timer_setup.timer_reference = 2;                // timer channel 2
    timer_setup.timer_mode = (TIMER_PERIODIC | TIMER_US_VALUE);  // single shot us timer
    timer_setup.timer_value = 100;                  // 100µ delay
    fnConfigureInterrupt((void *)&timer_setup);     // enter interrupt and start timer
}
```

This example is equivalent to that in the previous section but with `TIMER_SINGLE_SHOT` replaced by `TIMER_PERIODIC`.

A periodic timer can be stopped by calling the interface with the mode set to `TIMER_STOP`.

See Appendix C for further examples of generating a single-shot interrupt delay for various processor types and using various timer modules in the processors.

5. Configuring a Pulse-Width-Modulation Signal on a Timer Output Pin

It is often possible to generate PWM signals from general purpose timers. Some processors have, in addition, dedicated PWM modules optimised for this task.

The following example shows two PWM signals (CCP0 and CCP1) being generated from a single general purpose timer channel on a Luminary Micro device.

```
static void fnConfigure_Timer(void)
{
    static TIMER_INTERRUPT_SETUP timer_setup = {0};  // interrupt configuration parameters
    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.int_priority = PRIORITY_TIMERS;
    timer_setup.int_handler = 0;                 // no interrupt
    timer_setup.timer_reference = 0;             // timer channel 0
    timer_setup.timer_mode  = (TIMER_PWM_B);     // generate PWM signal on timer output port
    timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1000));// generate 1000Hz
    timer_setup.pwm_value   = 20;                // 20% PWM (high/low)
    fnConfigureInterrupt((void *)&timer_setup);// enable PWM signal
    timer_setup.timer_mode  = (TIMER_PWM_A | TIMER_DONT_DISTURB);
                                  // now set output A but don't disturb (reset) output B
    timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1500));// generate 1500Hz
    timer_setup.pwm_value   = 35;               // 35% PWM (high/low)
    fnConfigureInterrupt((void *)&timer_setup); // enable PWM signal
}
```

There is no interrupt involved with a PWM channel and the PWM output runs continuously until stopped.

The initialisation includes also the configuration of the port output for PWM use.

By recalling the initialisation but with different frequencies or PWM percentage values (0..100% in steps of 1%) changes to the present setting can be achieved. Whether the `TIMER_DONT_DISTURB` flag is used depends on whether a timer reset (takes place when called without the flag) is desired or not.

The following shows first one channel being stopped (the other will continue to operate) and then the second channel being disable. In this case, when both channels have been disabled, the timer channel will be set back to its power-down state to ensure lowest power consumption when not used.

```
static void fnStop_PWM(void)
{
    static TIMER_INTERRUPT_SETUP timer_setup = {0};  // interrupt configuration parameters
    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.timer_reference = 0;                 // timer channel 0
    timer_setup.timer_mode  = (TIMER_STOP_PWM_B | TIMER_DONT_DISTURB);
                                                     // stop B but don't disturb A
    fnConfigureInterrupt((void *)&timer_setup);      // disable PWM signal

    timer_setup.timer_mode  = (TIMER_STOP_PWM_A);    // stop A and power down timer module
    fnConfigureInterrupt((void *)&timer_setup);      // disable PWM signal
}
```

Note that once a PWM channel has been disabled its PWM port output state may not be defined (resulting in a continuous '0' or '1'). It may therefore be necessary to convert the port back to GPIO use by adding, for example, `_FLOAT_PORT(B, PORTB_BIT0);` (assuming the PWM output is on port B-0 and the input floating state is suitable). *The state after a channel power down may also be different to the case of simply disabling a module*.

See Appendix E for further examples of generating PWM signals for various processor types and using various timer modules in the processors.

6. Configuring a Timer with external Clock Input

In some cases it is necessary to count external pulses; for example in order to measure an external frequency, measure a pulse width, duty cycle or phase between two inputs. This mode of operation is normally referred to as capture mode.

The following example shows how an input can be configured on the SAM7X for use as clock and subsequently how the timer counter value is read.

The SAM7X has 3 individual 16 bit timers; there are several pins that can be used by the timer as outputs or inputs.

TCLK0, TCLK1, TCLK2 – these are inputs (called external inputs – these can be used as clock inputs)

TIOA0, TIOA1, TIOA2 – these can be inputs or outputs (called internal I/O signals – these can be used as clock inputs)

TIOB0, TIOB1, TIOB2 – these can be inputs or outputs (called internal I/O signals – these can be used as trigger inputs but not clock inputs)

The timer counter can be incremented on either the rising or falling edge of the signal.

When an external clock source is selected it can be XC0, XC1 or XC2, where these are sourced by the following possible combinations:

XC0 can be TCLK0, TIOA1 or TIOA2

XC1 can be TCLK1, TIOA0 or TIOA2

XC2 can be TCLK2, TIOA0 or TIOA1

The maximum frequency of an external signal is 2/5th the master clock.

The timers are flexible so this example is just one of various configurations – it simply configures the input as clock to the timer counter so that the counter is incremented at the frequency of the external signal. The counter runs from its initial value of 0x0000 up to a maximum value of 0xffff. After the value 0xffff is reached it overruns to 0x0000. By reading the timer counter value at a two instances in time the external frequency (assuming a stable pulse rate) can be measured. If the timer overflows, an interrupt on the overflow allows the timer width to be increased by incrementing a further variable (creating a 32 bit timer value).

```
static void fnConfigure_Timer(void)
{
    static TIMER_INTERRUPT_SETUP timer_setup = {0}; // interrupt configuration parameters
    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.int_priority = PRIORITY_TIMERS;
    timer_setup.int_handler = fnOverflow;           // interrupt handler on overflow
    timer_setup.timer_reference = 0;                // timer channel 0
    timer_setup.timer_mode  = (TIMER_SOURCE_TCLK0 | TIMER_SOURCE_RISING_EDGE);
                                                    // timer clock input and edge
    fnConfigureInterrupt((void *)&timer_setup);     // enable PWM signal
}
```

This example shows timer 0 being set to be clocked from the TCLK0 input, incremented on its rising edge. An interrupt handler is specified for timer counter overflows (a value of 0 for the interrupt handler lets the timer overflow without generating an interrupt).

The possible timer sources are (only one may be defined)

```
TIMER_SOURCE_TCLK0
TIMER_SOURCE_TCLK1
TIMER_SOURCE_TCLK2
TIMER_SOURCE_TIOA0
TIMER_SOURCE_TIOA1
TIMER_SOURCE_TIOA2
```

It is possible to define the same source for more than one timer. The user must however be aware that not all source combinations are possible – for example if TCLK0 and TCLK1 are used by two timers, the third timer cannot use TIOA2 (this is because the external signals XC0 and XC1 have been allocated and TIOA2 is not available via XC2. If this were attempted no clock would be connected. Furthermore, since the allocation of TIOA inputs can be from two possible XC sources the allocation priority is defined as:

1)TIOA0 will be taken from XC1, if available. If not it will be taken from XC2

2) TIOA1 will be taken from XC2, if available. If not it will be taken from XC0

3) TIOA2 will be taken from XC0, if available. If not it will be taken from XC1

Calling `fnConfigureInterrupt((void *)&timer_setup)` with `timer_setup.timer_mode = TIMER_ DISABLE;` will disable the timer (power down) and also disconnect its source. This pin will however be left configured as timer input so will need to be reconfigured if required for a different function afterwards.

Assuming that the overflow interrupt is incrementing a variable called usCounterOverflow a 32 bit timer value can be read by performing

```
ulCounter = (_COUNTER_VALUE(0) + (usCounterOverflow << 16));
```

This shows the macro `_COUNTER_VALUE()` which is used for direct timer counter register access.

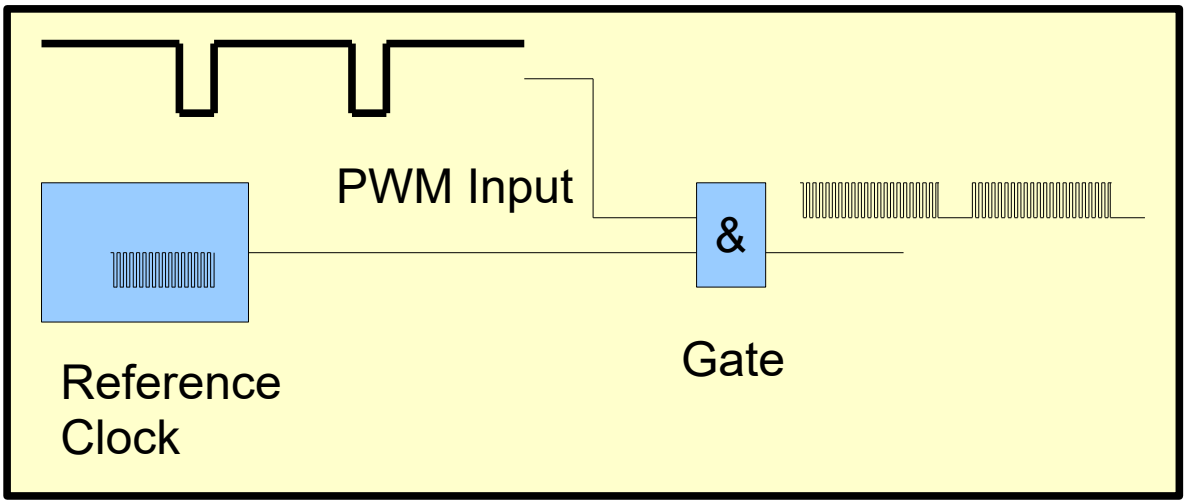
7. Measuring a PWM Input

Some sensors deliver their output value as a PWM signal. For example, 100% mark-space-ratio of a 20kHz square wave may represent 20mA output, 0% mark-space-ratio of the same frequency may represent 0mA and 50% mark-space-ratio 10mA. *The exact mark-space-ratio being proportional to the output value of the sensor's output range*.

The advantage of this solution is that it is digital and so robust when there is possible noise and interference; the frequency itself is not critical and can fluctuate because it is not the frequency but its mark-space that is of importance; the value repeats and so a measurement can be performed over multiple cycles to filter out any fluctuations or interference.

The possible disadvantage is that it may not always be simple to use a hardware time to perform the measurement.

The following represents a reference method of performing the measurement when there is a hardware timer available that includes a gated input to a counter. Afterwards a further technique is show that can be used by processors that allow DMA transfers to be triggered by edges on input pins (such as various Kinetis parts).



Consider the use of a simple AND gate in the diagram above. Using a reference clock on one input and the PWM signal to be measured on the other, the AND gate gates the reference clock through to its output only when the PWM signal is at a logic '1' state.

The reference clock frequency should be a lot higher than the PWM frequency so that it is easy to distinguish how many of its cycle are passed through each time the PWM input is high.

If the gated output is used as the input to a counter it counts the number of reference clock cycles that are passed through during a certain period of time and, since the number of periods of the reference clock are know during the measurement period, the PWM can be calculated by the formula

```
((Gated clock pulses during period / Reference clock pulses during
period) x 100)%
```

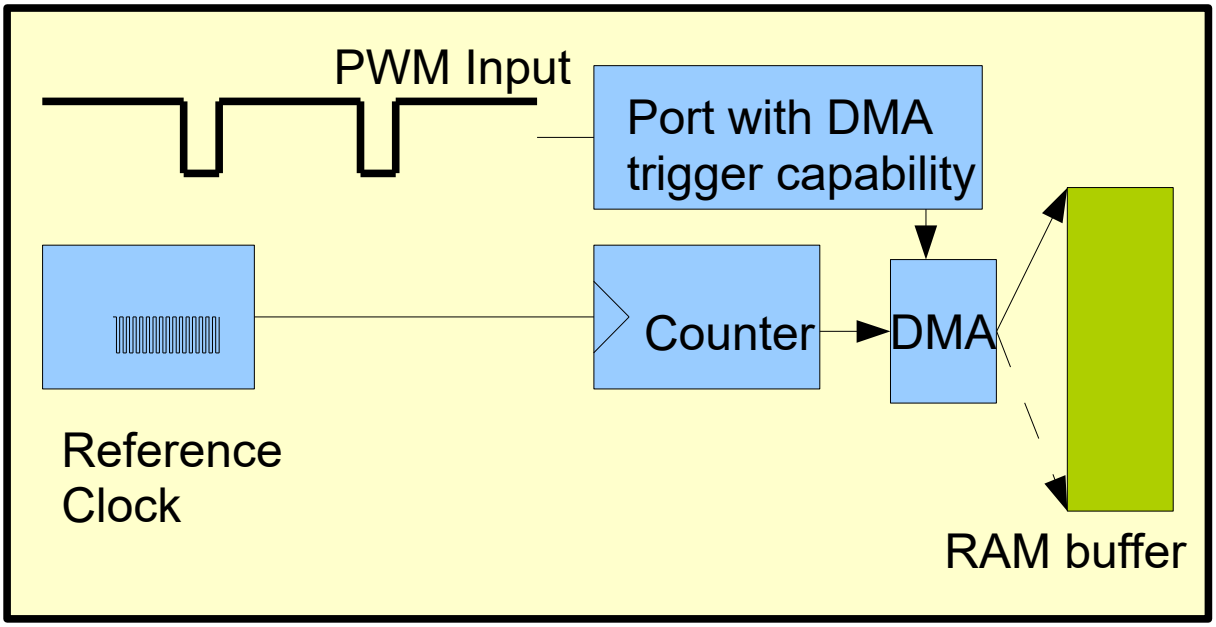
The measurement relies on the reference clock frequency being known accurately and also the measurement period being exact but can be improved to be less sensitive of the exact period if the reference clock pulses can be counted at the same time.

For highest accuracy the period of the measurement needs to be much longer than a PWM period and the reference frequency needs to be much higher than the PWM frequency. If it is possible to synchronise the start and top of the measurement period with a multiple of PWM cycles the accuracy is improved over shorter measurement periods.

The accuracy that is possible thus depends on the PWM frequency itself, the period that can be used for the measurement (effectively its sample frequency) and any tricks that the HW timer being used *may* allow to synchronise the measurement.

Note that when multiple PWM inputs are to be measured each one requires its own gate and counter.

The following PWM measurement illustrates how port triggered DMA transfers can allow a single HW counter to be used to measure a PWM signal with high accuracy.

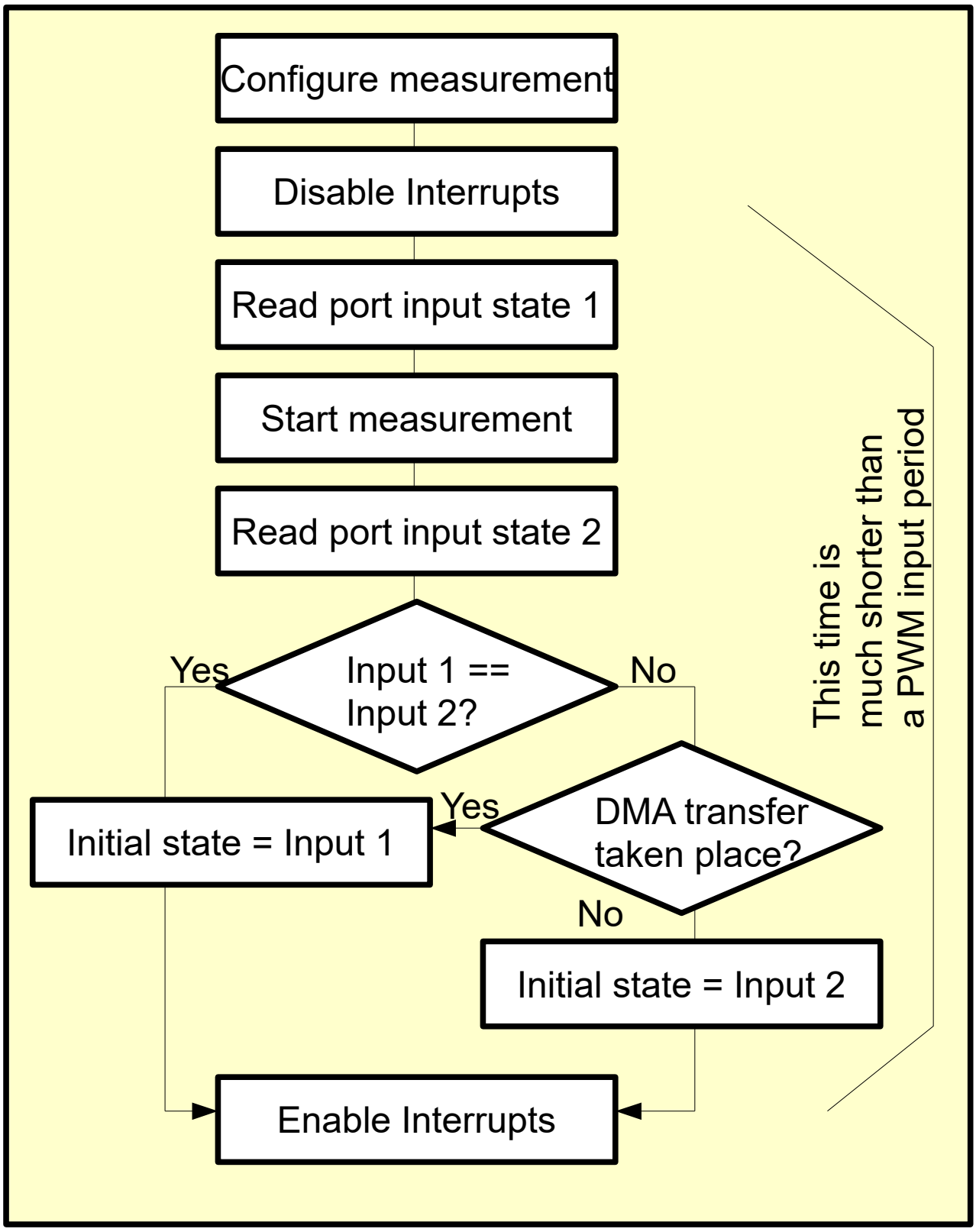


The port is configured to trigger a single DMA transfer on each input edge of the PWM input. Each trigger causes a transfer of the momentary reference clock counter value to a location in RAM, which increments after each transfer. The measurement is complete after a predefined number of transfers has been triggered or if a period expires (eg. when there is no PWM input frequency available or it is signalling 0% (continuous '0') or 100% (continuous '1').

After the period has expired the RAM buffer contains a number of time stamps (with reference clock resolution) for each PWM input '0' to '1' and '1' to '0' changes (edges). Based on the time stamps, the '1' and '0' durations can be calculated over one or more input cycles, whereby fast sampling is possible when just a small number of input periods are required.

There is one complication involved due to the fact that it is imperative to know whether the initial transfer was due to a falling or rising input edge, otherwise the PWM value calculated will be incorrect (99.9% mark-space-ratio could be misinterpreted as 0.1% if the polarity were not known accurately!).

A technique to allow the initial state to be reliably determined is shown in the flow diagram below:



As long as multiple ports can be used to trigger the same procedure on multiple DMA channels a single hardware counter can be shared by all measurements.

8. Input Capture

Hardware timers often have the ability to capture their present counter value when triggered by an external event, such as the falling edge on an input.

The following shows how channel 1 of FlexTimer/TPM in a Kinetis device is used to capture on each falling edge on its dedicated timer input pin (FTM0_CH1):

```
    static TIMER_INTERRUPT_SETUP timer_setup = {0};  // interrupt configuration parameters
    timer_setup.int_type = TIMER_INTERRUPT;
    timer_setup.int_priority = PRIORITY_TIMERS;
    timer_setup.int_handler = timer_int;
    timer_setup.timer_reference = 0;                 // timer 0
    timer_setup.capture_channel = 1;                 // channel 1
    timer_setup.capture_prescaler = 128;             // 1, 2, ..128 possible
    timer_setup.timer_mode = (TIMER_CAPTURE_FALLING);// capture interrupt on falling edge
    fnConfigureInterrupt((void *)&timer_setup);      // enter interrupt for timer test
```

In this example timer 0 free-runs, clocked from its defined source and with a pre-scale divider of 128. A capture takes place when a falling edge if detected on the timer input for channel 1, causing the timer's present count value to be latched to a dedicated internal register. A user interrupt call-back is also defined to be called by an interrupt resulting from this event; an example of such a callback is shown below:

```
static __callback_interrupt void timer_int(void)
{
    static volatile unsigned short usLastCapture = 0;
    usLastCapture = CAPTURE_VALUE(0, 1);             // update the last capture value
    TOGGLE_TEST_OUTPUT();
}
```

In this reference the present capture value is copied to a local variable and an output is toggled to better visualise the event taking place. The macro used to read the captured value from the internal capture register is hardware specific and in the case of the Kinetis parts it directly accesses the 16 bit capture register associated with channel 1 of FlexTimer 0.

9. Conclusion

This document has discussed various hardware timer uses, how frequencies and PWM signals can be generated, as well as the hardware timer interfaces in the µTasker project.

A section also discusses practical methods of using hardware time capabilities to measure PWM inputs.

Various processor specific details are included in the appendixes.

Modifications:

- V0.01 29.8.2009: First preliminary version with only Luminary Micro specific use
- V0.02 14.1.2010: Add SAM7X PWM details in appendix D
- V0.03 10.3.2010: Add external counter mode
- V0.04 1.1.2011: Add LM3Sxxxx PWM details in appendix D
- V0.05 2.02.2012: Add Kinetis PWM from FlexTimer in appendix D
- V0.06 6.05.2014: Add Kinetis and Coldfire V2 supported modules in appendix A and further examples of single-shot, period and PWM use for Freescale processors.
- V0.0710.2016: Add PWM measurement.
- V0.08 21.08.2018: Added interrupt call-back to Kinetis PWM mode reference.
- V0.09 27.08.2018: Added input capture chapter.

Appendix A – List or Processors and Timer Modules Supported

Kinetis

	Single-shot	Periodic	PWM	Notes
PIT (Periodic Interrupt Timer) 32 bit timers	Yes	Yes	No *(no output)*	KL devices have typically 2 PIT channels; K devices have typically 4 PIT channels.
FlexTimer 16 bit timers	Yes *(One per FlexTimer)*	Yes *(One per FlexTimer)*	Yes *(2 to 8 outputs for each FlexTimer module)*	K devices – usually there are 2 to 4 FlexTimers A single FlexTimer has between 2 and 8 channels. Frequency of all channels of each FlexTimer module are shared.
TPM 16 bit timer	Yes *(One per FlexTimer)*	Yes *(One per FlexTimer)*	Yes *(2 to 8 outputs for each FlexTimer module)*	KL devices (very similar to FlexTimer)

Coldfire V2

	Single-shot	Periodic	PWM	Notes
PIT (Periodic Interrupt Timer) 16 bit timers	Yes	Yes	No *(no output)*	2 to 4 PITs available, depending on device, whereby PIT0 is usually used by the µTasker OS Tick
DMA Timers 32 bit timers	Yes	Yes	No *(no output)*	4 DMA timers available
GPT 16 bit timer	No	No	No	4 channels Input capture function and can be used for positive or negative edge interrupt
PWM Module 8 bit timer	No	No	Yes	8 channels

Appendix B – Examples of Single-Shot Interrupt Delays

Kinetis K/KL PIT or Coldfire V2

```
PIT_SETUP pit_setup;                 // interrupt configuration parameters
pit_setup.int_type = PIT_INTERRUPT;
pit_setup.int_handler = test_timer_int;        // test a single shot timer
pit_setup.int_priority = PIT1_INTERRUPT_PRIORITY;
pit_setup.count_delay = PIT_US_DELAY(3245);               // 3245us delay
pit_setup.mode = PIT_SINGLE_SHOT;                      // one-shot interrupt
pit_setup.ucPIT = 1;                                             // use PIT1
fnConfigureInterrupt((void *)&pit_setup);   // enter interrupt for PIT1
```

Kinetis K/KL FlexTimer / TPM

```
TIMER_INTERRUPT_SETUP timer_setup;   // interrupt configuration parameters
timer_setup.int_type = TIMER_INTERRUPT;
timer_setup.int_priority = PRIORITY_TIMERS;
timer_setup.int_handler = timer_int;
timer_setup.timer_reference = 0;               // FlexTimer/TPM channel 0
timer_setup.timer_mode = (TIMER_SINGLE_SHOT);  // period timer interrupt
timer_setup.timer_value = TIMER_US_DELAY(100); // single-short 100us
fnConfigureInterrupt((void *)&timer_setup);    // enter and start timer
```

Coldfire V2 DMA Timer

```
DMA_TIMER_SETUP dma_timer_setup;    // interrupt configuration parameters
dma_timer_setup.int_type = DMA_TIMER_INTERRUPT;
dma_timer_setup.int_handler = DMA_timer_int;
dma_timer_setup.channel = 1;                          // DMA timer channel 1
dma_timer_setup.int_priority = DMA_TIMER1_INTERRUPT_PRIORITY;
                                                 // define interrupt priority
dma_timer_setup.mode = (DMA_TIMER_INTERNAL_CLOCK |
                                          DMA_TIMER_SINGLE_SHOT_INTERRUPT);
dma_timer_setup.count_delay = DMA_TIMER_US_DELAY(1,1,6345);
                                        // 6345us delay using no dividers
fnConfigureInterrupt((void *)&dma_timer_setup);  // enter and start timer
```

Appendix C – Examples of Periodic Interrupts

Kinetis K/KL PIT or Coldfire V2

```
PIT_SETUP pit_setup;                  // interrupt configuration parameters
pit_setup.int_type = PIT_INTERRUPT;
pit_setup.int_handler = test_timer_int;        // test a single shot timer
pit_setup.int_priority = PIT1_INTERRUPT_PRIORITY;
pit_setup.count_delay = PIT_MS_DELAY(25);               // 25ms interrupt
pit_setup.mode = PIT_PERIODIC;                        // periodic interrupt
pit_setup.ucPIT = 1;                                             // use PIT1
fnConfigureInterrupt((void *)&pit_setup);   // enter interrupt for PIT1
```

To stop a periodic PIT timer `pit_setup.mode = PIT_STOP;` can be used.

Kinetis K/KL FlexTimer / TPM

```
TIMER_INTERRUPT_SETUP timer_setup;   // interrupt configuration parameters
timer_setup.int_type = TIMER_INTERRUPT;
timer_setup.int_priority = PRIORITY_TIMERS;
timer_setup.int_handler = timer_int;
timer_setup.timer_reference = 1;               // FlexTimer/TPM channel 1
timer_setup.timer_mode = (TIMER_PERIODIC);     // period timer interrupt
timer_setup.timer_value = TIMER_MS_DELAY(150); // 150ms periodic interrupt
fnConfigureInterrupt((void *)&timer_setup);    // enter interrupt and start
```

To stop a periodic timer `timer_setup.mode = TIMER_STOP;` can be used.

Coldfire V2 DMA Timer

```
DMA_TIMER_SETUP dma_timer_setup;    // interrupt configuration parameters
dma_timer_setup.int_type = DMA_TIMER_INTERRUPT;
dma_timer_setup.int_handler = DMA_timer_int;
dma_timer_setup.channel = 2;                          // DMA timer channel 2
dma_timer_setup.int_priority = DMA_TIMER1_INTERRUPT_PRIORITY;
                                                 // define interrupt priority
dma_timer_setup.mode = (DMA_TIMER_INTERNAL_CLOCK |
                                               DMA_TIMER_PERIODIC_INTERRUPT);
dma_timer_setup.count_delay = DMA_TIMER_MS_DELAY(2,1,15);
                                          // 15ms delay using /2 pre-scaler
fnConfigureInterrupt((void *)&dma_timer_setup);  // enter and start timer
```

To stop a periodic DMA timer `dma_timer_setup.mode = PIT_STOP;` can be used.

Appendix D – Examples of Generating PWM Signals

AT91SAM7X – PWM

The SAM7X has a PWM controller with 4 channels. The outputs are called PWM0..PWM3 and each is available on two possible output pins.

In addition to the PWM controller, the general purpose timers can also be used to generate PWM signals. Only the PWM controller is discussed here.

To enable the PWM controller support in the µTasker project the define `SUPPORT_PWM_CONTROLLER` must be set.

The following is an example of using the PWM controller together with the SAM7X. It shows 4 PWM signals being generated, using all 4 PWM channels.

```
TIMER_INTERRUPT_SETUP timer_setup = {0};       // interrupt configuration parameters
timer_setup.int_type = PWM_CONFIGURATION;
timer_setup.timer_reference = 2;               // PWM channel 2
timer_setup.int_type = PWM_CONFIGURATION;
timer_setup.timer_mode  = (TIMER_PWM_ALT);     // configure PWM signal on alternative PWM2
                                                                                output
timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1000));// generate 1000Hz
                                                                     on timer output
timer_setup.pwm_value   = _PWM_PERCENT(20, TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1000)));
                                               // 20% PWM (high/low)
fnConfigureInterrupt((void *)&timer_setup);    // enter configuration for PWM test
timer_setup.timer_reference = 3;
timer_setup.timer_mode  = (TIMER_PWM);         // generate PWM signal on PWM3 output and
                                                              synchronise all PWM outputs
timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1500));
timer_setup.pwm_value   = _PWM_TENTH_PERCENT(706,
                                            TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1500)));
                                             // 70.6% PWM (high/low) on different channel
fnConfigureInterrupt((void *)&timer_setup);   // enter configuration for PWM test
timer_setup.timer_reference = 0;
timer_setup.timer_mode  = (TIMER_PWM_ALT);    // generate PWM signal on PWM0
timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(2000));
timer_setup.pwm_value   = _PWM_TENTH_PERCENT(995,
                                            TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(2000)));
                                             // 99.5% PWM (high/low) on different channel
fnConfigureInterrupt((void *)&timer_setup);   // enter configuration for PWM test
timer_setup.timer_reference = 1;
timer_setup.timer_mode  = (TIMER_PWM | TIMER_PWM_START_0 | TIMER_PWM_START_1 |
                           TIMER_PWM_START_2 | TIMER_PWM_START_3);
                     // generate PWM signal on PWM1 output and synchronise all PWM outputs
timer_setup.timer_value = TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(3000));
timer_setup.pwm_value   = _PWM_TENTH_PERCENT(25,
                                            TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(3000)));
                                            // 2.5% PWM (high/low) on different channel
fnConfigureInterrupt((void *)&timer_setup);  // enter configuration for PWM test
```

Note the following points:

1) Since the PWM controller is being used, and not a general purpose timer, the `int_type` is set to `PWM_CONFIGURATION`. The `TIMER_INTERRUPT_SETUP` is otherwise used as for timer control.

2) `timer_reference` is used to specify the PWM controller channel (0..3). In the example all 4 channels are configured.

3) The primary output pin is used when the mode is set to `TIMER_PWM` and the alternative output is used when `TIMER_PWM_ALT` is set. The primary outputs are PB19, PB20, PB21, PB22 of the PWM controller channels 0, 1, 2 and 3.
The secondary outputs are PB27, PB28, PB29 and PB30 for the PWM controller channels 0, 1, 2 and 3.

4) Although the 4 channels are configured independently, their counters are not actually started until the final channel is configured. All 4 are started using `TIMER_PWM_START_0 | TIMER_PWM_START_1 | TIMER_PWM_START_2 | TIMER_PWM_START_3`, which has the effect of synchronising all 4 channels.

5) To stop channels from operating the mode flags TIMER_PWM_STOP_0, TIMER_PWM_STOP_1 , TIMER_PWM_STOP_2 and/or TIMER_PWM_STOP_3 can be used. Should no further channels be running after this command the PWM controller will be powered down. In the powered down state the settings are however retained in the PWM module in the SAM7X.

LM3Sxxxx – PWM

The LM3Sxxxx has optional PWM generators. Each available one has two channel outputs which are named internally channel A and B for each generator. Externally the naming of the PWM outputs counts from 0, 1 to the highest one. For example, a device with 2 PWM generators will have outputs PWM0, PWM1, PWM2 and PWM3, whereby PWM0 and PWM are channels 0 and 1 of the first generator module and PWM2 and PWM3 are the channels 0 and 1 of the second PWM generator module.

In addition to the PWM controller, the general purpose timers can also be used to generate PWM signals. Only the PWM controller is discussed here.

To enable the PWM controller support in the µTasker project the define `SUPPORT_PWM_CONTROLLER` must be set.

The following is an example of using the PWM controller together with the LM3S8962. It shows 2 PWM signals being generated, using 2 PWM from 2 generator modules. The LM3S8962 has 3 PWM modules and can thus generate up to 6 PWM output signals at the same time.

```
TIMER_INTERRUPT_SETUP timer_setup = {0};       // interrupt configuration parameters
timer_setup.int_type = PWM_CONFIGURATION;
timer_setup.timer_reference = 2;               // PWM channel 2
timer_setup.int_type = PWM_CONFIGURATION;
timer_setup.timer_mode  = PWM_DIV_1;           // don't start yet
timer_setup.timer_value = PWM_FREQUENCY_VALUE(1000, 1); // generate 1000Hz on timer output
                                                          using PWM clock without divide
timer_setup.pwm_value   = _PWM_PERCENT(20, PWM_FREQUENCY_VALUE(1000, 1));
                                               // 20% PWM (high/low)
fnConfigureInterrupt((void *)&timer_setup);    // enter configuration for PWM test
timer_setup.timer_reference = 5;
timer_setup.timer_mode  = (TIMER_PWM_START_2 | TIMER_PWM_START_5 | PWM_DIV_1);
                                               // generate PWM signal on these outputs
timer_setup.timer_value = PWM_FREQUENCY_VALUE(1500, 1);
timer_setup.pwm_value   = _PWM_TENTH_PERCENT(706, PWM_FREQUENCY_VALUE(1500, 1));
                                              // 70.6% PWM (high/low) on different channel

fnConfigureInterrupt((void *)&timer_setup);   // enter configuration for PWM test
```

Note the following points:

1) Since the PWM controller is being used, and not a general purpose timer, the `int_type` is set to `PWM_CONFIGURATION`. The `TIMER_INTERRUPT_SETUP` is otherwise used as for timer control.
2) `timer_reference` is used to specify the PWM controller channel (0..5). In the example 2 channels (2 and 5) are configured.
3) The PWM outputs PWM2 and PWM5 are fixed on dedicated outputs, which may change with different parts. The driver code selects the appropriate peripheral function for the tested part but this needs to be verified for other parts in case they need a dedicated port configuration to be added.
4) Although the 2 channels are configured independently, their operation is not actually started until the final channel is configured. All 2 are started using `TIMER_PWM_START_2 | TIMER_PWM_START_5`, which has the effect of (roughly) synchronising all 4 channels.

5) To stop channels from operating the mode flags TIMER_PWM_STOP_2 and/or TIMER_PWM_STOP_5 can be used. Should no further channels be running after this command the PWM controller will be powered down to save energy.

6) Since PWM0 and PWM1 (PWM2 and PWM3, etc.) are derived from the same PWM generator with a single 16 bit counter the output frequency must be the same for these two outputs. The PWM mark/space ration can however be configured independently. The two outputs are always syhchronised since they are derived from the same counter.

7) PWM outputs not belonging to the same PWM generator are free to have different frequencies. The driver doesn't synchronise the output signals between PWM generators although the PWM controller in the LM3Sxxxx has some global synchronisation capabilities.

8) The PWM generator has a high degree of flexibility as to how the PWM signals are generated. The driver uses one fixed method as follows to generate the signals:
Initially the PWM output signal is at logical level '0'.
The PWM counter is loaded with the base frequency value and remains at '0' and counts down until the count value matches the PWM value for the specific channel (A or B), at which moment the output is set to logical '1'.
The counter continues to count down until it reaches the value 0x0000, at which moment it is automatically reloaded with the periodic value and the output is reset to logical '0' again. This results in the configured PWM mark/space value with the '1' phase occurring after the ''0 phase (right-aligned).

9) When configuring the PWM frequency and mark/space % value a divider is also specified. This is a divider to the PWM module which must be the same for all PWM generators and channels used at the same time. It can have the values 1, 2, 4, 8, 16, 32 or 64, which must also be specified in the mode setting (PWM_DIV_1, PWM_DIV_2, PWM_DIV_4, PWM_DIV_8, PWM_DIV_16, PWM_DIV_32 or PWM_DIV_64) – if nothing is specified PWM_DIV_1 is valid.
The PWM generators are therefore clocked by the system clock divided by the PWM divide value; a divide value of 1 gives the highest frequency and PWM resolutions; higher divide values allow slower signals to be generated and slightly reduced power consumption.

Kinetis – PWM

The FlexTimers in the Kinetis can generate edge-aligned or centre-aligned PWM outputs on each of their channels. In KL devices the *TPM is used instead but the interface is compatible*.

To enable the PWM controller support in the µTasker project the defines `SUPPORT_TIMER` and `SUPPORT_PWM_MODULE` must be set, whereby the FlexTimer is used and not a specific PWM module.

The following is an example of using the PWM controller together with the Kinetis. It shows 2 PWM signals being generated by FlexTimer 0.

```
PWM_INTERRUPT_SETUP pwm_setup;
pwm_setup.int_type    = PWM_INTERRUPT;
pwm_setup.pwm_mode    = (PWM_SYS_CLK | PWM_PRESCALER_16); // clock PWM timer from the
                                                         system clock with /16 pre-scaler
pwm_setup.int_handler = 0;                              // no interrupt call-back
pwm_setup.pwm_reference = (_TIMER_0 | 3);               // timer module 0, channel 3
pwm_setup.pwm_frequency = PWM_TIMER_US_DELAY(TIMER_FREQUENCY_VALUE(1000), 16);
                                                        // generate 1000Hz on PWM output
pwm_setup.pwm_value   = _PWM_PERCENT(20, pwm_setup.pwm_frequency); // 20% PWM (high/low)
fnConfigureInterrupt((void *)&pwm_setup);               // enter configuration for PWM test
pwm_setup.pwm_reference = (_TIMER_0 | 2);               // timer module 2, channel 2
pwm_setup.pwm_mode |= PWM_POLARITY;                     // change polarity of second channel
pwm_setup.pwm_value   = _PWM_TENTH_PERCENT(706, pwm_setup.pwm_frequency);
                                                  // 70.6% PWM (low/high) on different channel
fnConfigureInterrupt((void *)&pwm_setup);
```

Note the following points:

1) The clock source can be from the SYSCLK (as show in the example) or from an external clock source (`PWM_EXTERNAL_CLK` instead of `PWM_SYS_CLK`). When an external clock is used thevalues passed for the frequency and PWM will depend on that frequency and so the calculation macros cannot be used.
   The clock input used is either from FTM_CLKIN0 or FTM_CLKIN1 depending on the project define `FTM_CLKIN_1`. Care should be taken when using an external clock since the clock pins are multiplexed with the main crystal/clock pins.
2) The behaviour of the Flex Timer counter and its outputs during debugging (BDM mode) can be defined by the selection of the define `FTM_DEBUG_BEHAVIOUR` in `app_hw_kinetis.h`. It can be allowed to run or stopped and its outputs can be frozen, continue running or be held in a defined state.
3) All PWM channel outputs from a Flex Timer share the same clock and PWM period. Only the PWM mark-space values of each can be changed along with polarity.
4) All PWM channels on a single Flex Timer also share the same mode (edge or centre aligned).
5) All PWM outputs on a single Flex Timer are synchronised according to edge or centre alignment mode).
6) To disable all PWM outputs the function can be called using `pwm_setup.pwm_mode = 0;`
7) An interrupt will be generated on each cycle when `pwm_setup.int_handler` is not 0 but instead set to a user interrupt call-back function. In this case also `pwm_setup.int_priority` should be assigned a suitable interrupt priority.

The following diagram shows the effect of the polarity and alignment options: